Fixing Non-determinism

Alexander Vandenbroucke Tom Schrijvers Frank Piessens

{alexander.vandenbroucke, tom.schrijvers, frank.piessens}@kuleuven.be

Department of Computer Science
KU Leuven

ABSTRACT

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

CCS Concepts

•Software and its engineering → Functional languages; Recursion;

Keywords

Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. INTRODUCTION

Non-determinism [24] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \; ? \; 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list $[1, 2]$. Now consider the next example:

$$\begin{aligned} &swap\ (m, n) = (n, m) \\ &pair = (1, 2) \; ? \; swap\ pair \end{aligned}$$

© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

The corresponding Haskell code is:

```
swap :: [(a, b)] → [(b, a)]
swap e = [(m, n) | (n, m) ← e]
pair :: [(Int, Int)]
pair = [(1, 2)] ++ swap pair
```

This is an executable model (we use ⋙ to denote the prompt of the GHCi Haskell REPL):

```
⋙ pair
[(1, 2), (2, 1), (1, 2), (2, 1) ...
```

We get an infinite list, although only two distinct outcomes ($(1, 2)$ and $(2, 1)$) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of $pair$ is then the set $\{(1, 2), (2, 1)\}$. We can execute this model: [1]

```
swap :: (Ord a, Ord b) ⇒ Set (a, b) → Set (b, a)
swap = map (λ(m, n) → (n, m))
pair :: Set (Int, Int)
pair = singleton (1, 2) `union` swap pair
⋙ pair
fromList *** Exception:  <loop>
```

Haskell lazily prints the first part of the Set constructor, and then has to compute $union$ infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution $\{(1, 2), (2, 1)\}$.

This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple

[1] Here *map* comes from `Data.Set`

graph problem, which could not be solved with the more specific semantics.

- We provide a set of benchmarks to demonstrate the expressivity of our approach and evaluate the performance of our implementation.

2. OVERVIEW

2.1 Non-determinism

A computation is non-deterministic if it has several possible outcomes. In this paper, we interpret such non-deterministic computations as the set of their results.

Consider the following non-deterministic expression that produces either 1 or 2:

$$1 \; ? \; 2$$

For this example, the semantics is given by the set $\{1, 2\}$. The semantic function $[\![\cdot]\!]$ formally characterizes this interpretation:

$$[\![n]\!] = \{n\}$$
$$[\![e_1 \; ? \; e_2]\!] = [\![e_1]\!] \cup [\![e_2]\!]$$

The semantics of a literal n is the singleton of n, and the semantics of a choice $e_1 \; ? \; e_2$ is the union of the semantics of the left and right branches. A simple calculation shows that $[\![1 \; ? \; 2]\!]$ is indeed $\{1, 2\}$.

$$[\![1 \; ? \; 2]\!] = [\![1]\!] \cup [\![2]\!] = \{1\} \cup \{2\} = \{1, 2\}$$

Let us extend our semantics to allow for addition of non-deterministic values:

$$[\![e_1 + e_2]\!] = \{n + m \mid n \in [\![e_1]\!], m \in [\![e_2]\!]\}$$

Now, consider the expression:

$$(1 \; ? \; 2) + (1 \; ? \; 2)$$

Here we have an expression that contains an addition of two choices. This expression has 4 possible outcomes of which two coincide: the result is either 2 (1+1), 3 (1+2 or 2+1) or 4 (2+2).[2] Again calculation gives the expected result:

$$\begin{aligned} &[\![(1 \; ? \; 2) + (1 \; ? \; 2)]\!] \\ &= \{n + m \mid n \in [\![1 \; ? \; 2]\!], m \in [\![1 \; ? \; 2]\!]\} \\ &= \{n + m \mid n \in \{1, 2\}, m \in \{1, 2\}\} \\ &= \{2, 3, 4\} \end{aligned}$$

Recursion The next example presents a recursive non-deterministic expression *pair* which chooses between $(1, 2)$ and the new primitive *swap* to swap *pair*'s components around.

$$pair = (1, 2) \; ? \; swap \; pair$$

The semantics of $swap \; e$ is the set obtained by flipping all pairs in the set of results of e:

$$[\![swap \; e]\!] = \{(m, n) \mid (n, m) \in [\![e]\!]\}$$

The semantics of *pair* are given by the following recursive equation:

$$\begin{aligned} & [\![pair]\!] = [\![(1, 2) \; ? \; swap \; pair]\!] \\ \iff \; & [\![pair]\!] = [\![(1, 2)]\!] \cup [\![swap \; pair]\!] \\ \iff \; & [\![pair]\!] = \{(1, 2)\} \cup \{(m, n) \mid (n, m) \in [\![pair]\!]\} \quad (1) \end{aligned}$$

[2]The conventional list-based semantics would be $[2, 3, 3, 4]$.

This equation admits infinitely many solutions, e.g.

$$\begin{aligned} &[\![pair]\!] = \{(1, 2), (2, 1)\}, \\ &[\![pair]\!] = \{(0, 0), (1, 2), (2, 1)\}, \\ &[\![pair]\!] = \{(1, 1), (1, 2), (2, 1)\}, \\ &\ldots \end{aligned}$$

However, we can identify a *least* solution: the set $\{(1, 2), (2, 1)\}$ is contained in every other solution.

As we saw previously, a naive translation of this idea in Haskell *does not work*:

```
det :: Ord a ⇒ a → Set a
det x = singleton x
(?) :: Ord a ⇒ Set a → Set a → Set a
a ? b = union a b
swap :: (Ord a, Ord b) ⇒ Set (a, b) → Set (b, a)
swap = Data.Set.map (λ(x, y) → (y, x))
pair :: Set (Int, Int)
pair = det (1, 2) ? swap pair
 ⋙ pair
fromList *** Exception:  <loop>
```

The reason it does not work is that under Haskell's (cpo-based) semantics the equation (1) has a different least solution: $\bot$ (i.e. non-termination).[3] As this additional $\bot$ in the domain is clearly not the desired solution, we cannot rely on Haskell's native semantics for recursion.

The main contribution of this paper is to reformulate the problem as a *different* least fixed point problem, for which we can iteratively compute the solution. Moreover, our approach incurs minimal overhead for the programmer, compared to writing the function using the conventional recursive approach.

2.2 Effect Handlers

Monads are a way to model side-effects such as non-determinism in a pure functional programming language [25]. In this paper, we use effect handlers to construct a monad for non-determinism and recursion. Effect handlers [10, 27] factor the problem of modeling effectful computations into two parts: first a syntax is introduced to represent all relevant operations, second effect handlers are defined that interpret the syntax within a semantic domain.

The syntax of non-deterministic computations can be modeled with the following data type ND, which supports three operations: $Success_{ND} \; a$ is a deterministic computation with result a, $Fail$ is a failed computation, and $Or \; l \; r$ represents a non-deterministic choice between two non-deterministic computations l and r.

```
data ND a
  = Success_ND a
  | Fail_ND
  | Or_ND (ND a) (ND a)
```

Because the above data type is a free monad [20] we can easily define the following *Monad* instance:

```
instance Monad ND where
  return a = Success_ND a
  Success_ND a >>= f = f a
  Fail_ND      >>= f = Fail_ND
  Or_ND l r    >>= f = Or_ND (l >>= f) (r >>= f)
```

[3]In Haskell we work in the domain $\langle \mathcal{P}(\mathbb{N} \times \mathbb{N}) \cup \{\bot\}, \sqsubseteq \rangle$, where every type is inhabited by $\bot$, representing non-termination, and $\bot \sqsubseteq v$ for any value v.

This monad instance substitutes $f\ a$ for every $Success_{ND}\ a$ in the data structure, leaves *Fail* untouched, and recurses on both branches of $Or\ l\ r$. With this monad instance, the example $(1\ ?\ 2) + (1\ ?\ 2)$ is expressed as:

$$
\begin{aligned}
&exampleND :: ND\ Int\\
&exampleND = \mathbf{do}\ x \leftarrow return\ 1\ `Or_{ND}`\ return\ 2\\
&\qquad\qquad\ \ y \leftarrow return\ 1\ `Or_{ND}`\ return\ 2\\
&\qquad\qquad\ \ return\ (x + y)
\end{aligned}
$$

Values of type $ND\ a$ are abstract syntax trees. The function $exampleND$ constructs such an abstract syntax tree. The interpreter nd decodes this tree according to the semantics defined by $[\![\cdot]\!]$. It turns *Success* into a singleton set, *Fail* into an empty set, and Or into the union of the interpretations of its branches.

$$
\begin{aligned}
&nd :: Ord\ a \Rightarrow ND\ a \to Set\ a\\
&nd\ (Success_{ND}\ a) = singleton\ a\\
&nd\ Fail_{ND} \qquad\ = empty\\
&nd\ (Or_{ND}\ l\ r) \quad = union\ (nd\ l)\ (nd\ r)\\
&\\
&\ggg nd\ exampleND\\
&fromList\ [2,3,4]
\end{aligned}
$$

The equivalent of *pair* in the abstract syntax is *pairND*:

$$
\begin{aligned}
&pairND :: ND\ (Int, Int)\\
&pairND = return\ (1,2)\ `Or_{ND}`\ fmap\ swap\ pairND\ \mathbf{where}\\
&\quad swap\ (x,y) = (y,x)\\
&\\
&\ggg nd\ pairND\\
&fromList\ ...
\end{aligned}
$$

Encoding the non-deterministic syntax as a data type does not solve the problem of non-termination. This can be explained by looking at the syntax tree for *pairND* in Figure 1: it is an infinite tree with an Or node at the root, a *Success* node in the left branch, and in the right branch another tree with Or at the root after which the pattern repeats. Obviously the interpreter nd does not terminate when interpreting such an infinite tree.

We have little chance of processing the infinite tree in a finite time if we do not represent it in a finite way. Yet the current recursive form of *pairND* hides the recursive call in the function body, making the construction of an infinite tree unavoidable.

3. EXPLICATING RECURSION

In order to obtain a finite syntax tree, we have to once more change the representation of non-deterministic computations. First, we add a constructor to the abstract syntax to explicitly represent a recursive call. Second, we replace all recursive calls with this constructor, and finally, we define a new effect handler that interprets the now finite syntax tree, producing the desired solution. The following data type models recursive calls in addition to non-determinism:

$$
\begin{aligned}
&\mathbf{data}\ NDRec\ i\ o\ a\\
&\quad = Success\ a && -(1)\\
&\quad \mid Fail && -(2)\\
&\quad \mid Or\ (NDRec\ i\ o\ a)\ (NDRec\ i\ o\ a) && -(3)\\
&\quad \mid Rec\ i\ (o \to NDRec\ i\ o\ a) && -(4)
\end{aligned}
$$

The first three constructors (1–3) capture non-determinism, exactly like the previously defined data type ND. The last constructor (4) captures a recursive call: $Rec\ a\ k$ represents a recursive call with argument $a :: i$, and continuation $k :: o \to NDRec\ i\ o\ a$. For convenience, we define four additional smart constructors. The smart constructor *rec* performs a recursive call and immediately wraps the result in a successful computation:

$$
\begin{aligned}
&rec :: i \to NDRec\ i\ o\ o\\
&rec\ i = Rec\ i\ Success
\end{aligned}
$$

The smart constructor *choice* picks a computation from a list in a non-deterministic fashion.

$$
\begin{aligned}
&choice :: [NDRec\ i\ o\ a] \to NDRec\ i\ o\ a\\
&choice = foldr\ Or\ Fail
\end{aligned}
$$

The smart constructor *choose* picks an element from a list in a non-deterministic fashion.

$$
\begin{aligned}
&choose :: [a] \to NDRec\ i\ o\ a\\
&choose = choice \circ map\ Success
\end{aligned}
$$

The smart constructor *guard* returns $()$ if its argument is true and fails otherwise.

$$
\begin{aligned}
&guard :: Bool \to NDRec\ i\ o\ ()\\
&guard\ b = \mathbf{if}\ b\ \mathbf{then}\ return\ ()\ \mathbf{else}\ Fail
\end{aligned}
$$

The data type *NDRec* is again a free monad, the corresponding monad instance is:

$$
\begin{aligned}
&\mathbf{instance}\ Monad\ (NDRec\ i\ o)\ \mathbf{where} && -(1)\\
&\quad return\ a = Success\ a && -(2)\\
&\quad Success\ a \gg\!= f = f\ a && -(3)\\
&\quad Fail \qquad \gg\!= f = Fail && -(4)\\
&\quad Or\ l\ r \quad \gg\!= f = Or\ (l \gg\!= f)\ (r \gg\!= f) && -(5)\\
&\quad Rec\ i\ k \quad \gg\!= f = Rec\ i\ (\lambda x \to k\ x \gg\!= f) && -(6)
\end{aligned}
$$

Lines 1–5 are identical to the $Monad$ instance for ND. Line (6) defines that $\gg\!=$ of a recursive call is obtained by a new recursive call to the same argument, but with an extended continuation. [4]

As an example, consider this latest incarnation of *pair*, written in the *NDRec* syntax:

$$
\begin{aligned}
&pairNDRec :: () \to NDRec\ ()\ (Int, Int)\ (Int, Int)\\
&pairNDRec\ () = return\ (1,2)\ `Or`\ \mathbf{do}\ (x,y) \leftarrow rec\ ()\\
&\qquad\qquad\qquad\qquad\qquad\qquad return\ (y,x)
\end{aligned}
$$

We interpret this program in the next section.

4. EFFECT HANDLER FOR EXPLICIT RECURSION

4.1 Denotational semantics

In this section we formalize the meaning of the abstract syntax. The meaning of a non-deterministic function f of type[5] $I \to NDRec\ I\ O\ O$, mapping I onto $NDRec\ I\ O\ O$ is given by a function $[\![\, f \,]\!] : I \to \mathcal{P}(O)$. This function maps values of type I onto a subset of the values described by the type O.

$$[\![\cdot]\!] : (I \to NDRec\ I\ O\ O) \to (I \to \mathcal{P}(O))$$

Let us first define the semantics of an easier case: suppose we already have an environment $s : I \to \mathcal{P}(O)$ that contains a partial set of solutions for every call $f(a)$, i.e.

$$s(a) \subseteq [\![\, f \,]\!](a) \text{ for every } a \in I$$

[4]The new continuation is the Kleisli composition $k \gg\!\!> f$ where $(\gg\!\!>) :: Monad\ m \Rightarrow (a \to m\ b) \to (b \to m\ c) \to (a \to m\ c)$.

[5]In deference to mathematical convention, we use uppercase characters for meta-variables I and O when using mathematical syntax.

Then for every syntax tree $t : ND\ I\ O\ O$ we can define a semantic function $\mathcal{R}[\![\, t \,]\!](s)$. This semantic function gives us the set of results associated with t, given s. Because the $ND\ I\ O\ O$ data type is inductive, we define the function $\mathcal{R}[\![\, \cdot \,]\!]$ using structural recursion, as follows:

$$\mathcal{R}[\![\, \cdot \,]\!] : NDRec\ I\ O\ O \to (I \to \mathcal{P}(O)) \to \mathcal{P}(O)$$

$$\mathcal{R}[\![\, Success\ x \,]\!](s) = \{x\} \tag{2}$$

$$\mathcal{R}[\![\, Fail \,]\!](s) = \emptyset \tag{3}$$

$$\mathcal{R}[\![\, Or\ l\ r \,]\!](s) = \mathcal{R}[\![\, l \,]\!](s) \cup \mathcal{R}[\![\, r \,]\!](s) \tag{4}$$

$$\mathcal{R}[\![\, Rec\ i\ k \,]\!](s) = \bigcup_{x \in s(i)} \mathcal{R}[\![\, k(x) \,]\!](s) \tag{5}$$

The two base cases are fairly simple: in the deterministic case (2) the result is just a singleton set of the result, and in the failure case (3) it is the empty set.

There are two inductive cases as well: a binary choice (4) and a recursive call (5). A binary choice is handled by taking the union of the results of the left and right branches. A recursive call has an argument i and a continuation k. The result is obtained by finding the set of outcomes in the environment s, and then applying the continuation k to every element x in $s(i)$, and taking the union of the result.

Now we can of define $[\![\, \cdot \,]\!]$ as follows: since $\mathcal{R}[\![\, f(a) \,]\!]([\![\, f \,]\!])$ gives the set of outcomes of $f(a)$ given environment $[\![\, f \,]\!]$, and $[\![\, f \,]\!](a)$ gives the set of outcomes of $f(a)$, the following must hold about $\mathcal{R}[\![\, f(a) \,]\!]([\![\, f \,]\!])$:

$$\begin{aligned}
&\mathcal{R}[\![\, f(a) \,]\!]([\![\, f \,]\!]) = [\![\, f \,]\!](a)\ \ (\forall a \in I) \\
&\quad [\text{ equivalence of } \lambda\text{-abstraction and } \forall\text{-quantification }] \\
\iff &\lambda a.\mathcal{R}[\![\, f(a) \,]\!](s) = \lambda a.[\![\, f \,]\!](a) \\
\iff &\lambda a.\mathcal{R}[\![\, f(a) \,]\!]([\![\, f \,]\!]) = [\![\, f \,]\!] \ [\ \eta\text{-reduction }] \\
\iff &[\![\, f \,]\!] \text{ is a fixed point of } \lambda s.\lambda a.\mathcal{R}[\![\, f(a) \,]\!](s)
\end{aligned}$$

Now all that remains is to choose a canonical fixed point for $[\![\, f \,]\!]$, such that it corresponds to the desired meaning. Note that environments of the type $I \to \mathcal{P}(O)$ (such as $[\![\, f \,]\!]$) are ordered by the ordering relation $\sqsubseteq$:

$$f \sqsubseteq g \iff \forall a \in I : f(a) \subseteq g(a)$$

The desired fixed point is *the least fixed point*[6] denoted by $lfp(\cdot)$, when fixed points are ordered by $\sqsubseteq$:

$$[\![\, f \,]\!] = lfp(\lambda s.\lambda a.\mathcal{R}[\![\, f(a) \,]\!](s)) \tag{6}$$

To make this more concrete, consider the denotational semantics of $pairNDRec$.

$$\begin{aligned}
&[\![\, pairNDRec\ () \,]\!](\,()\,) \\
&= lfp(\lambda s.\lambda().\mathcal{R}[\![\, pairNDRec\ () \,]\!](s))\big(()\big) \qquad [\text{ by (6) }] \\
&[\ \lambda().\{(1,2),(2,1)\} \text{ is the least fixed point }] \\
&= \big(\lambda().\{(1,2),(2,1\}\ ()\big) \\
&[\text{ function application }] \\
&\{(1,2),(2,1)\}
\end{aligned}$$

[6]The least fixed point is well defined, as the function is always continuous (if the syntax tree is finite) and therefore monotonic, but the domain $I \to \mathcal{P}(O)$ may not possess the finite ascending chain property, preventing us from computing the solution in a finite amount of time.

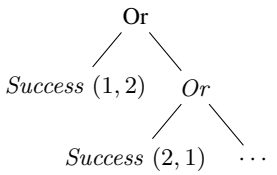

Figure 1: Infinite syntax tree created by $pairND$.

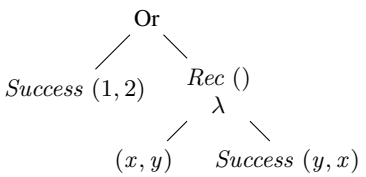

Figure 2: Finite syntax tree created by $pairNDRec\ ()$.

We can see that $\lambda().\{(1,2),(2,1)\}$ is the least fixed point because it is a fixed point and it is contained in every other fixed point.

4.2 Effect Handler Implementation

In this section we provide a Haskell implementation that correspond to the denotational semantics from the previous section. This implementation comes in the form of an effect handler [10, 27]. Like the denotational semantics, the effect handler is split into two parts: a part that delivers the solution for the entire function by computing a least fixed point (similar to $[\![\, \cdot \,]\!]$), and the counterpart of $\mathcal{R}[\![\, \cdot \,]\!]$, which, given an environment, computes the results for one syntax tree.

The first part is called $runNDRec$ and the second part is called go. The effect handler first computes the least fixed point of the $step$ function. The $step$ function is a function that takes an environment (of type $Map\ i\ (Set\ o)$ from `Data.Map`[7]) and obtains a new environment by running go for every entry in the environment. The fixed point of this function is again an environment, in which we look up i_0, the second argument to $runNDRec$.

```
runNDRec :: (Ord i, Ord o)
         ⇒ (i → NDRec i o o) → i → Set o
runNDRec expr i₀ = lfp step s₀ ! i₀ where
  s₀ = M.singleton i₀ empty
  step m = foldr (λk → go k (expr k)) m (M.keys m)
```

The function go, like $\mathcal{R}[\![\, \cdot \,]\!]$ proceeds by case analysis on the syntax tree. However, unlike $\mathcal{R}[\![\, \cdot \,]\!]$, go updates an environment instead of just returning a set of solutions. This difference is mainly for programming convenience.

```
go :: (Ord i, Ord o)
   ⇒ i → NDRec i o o
   → M.Map i (Set o) → M.Map i (Set o)
go i (Success a) m = M.insertWith union i (singleton a) m
go i Fail        m = m
go i (Or l r)    m = go i r (go i l m)
go i (Rec j k)   m = case M.lookup j m of
  Nothing → M.insert j empty m
  Just s  → foldr (go i ∘ k) m (toList s)
```

[7]Functions and types residing in `Data.Map` are imported with the qualifier `M`, except the lookup operator "!".

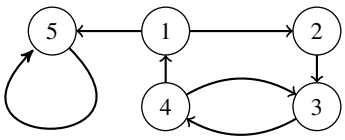

Figure 3: The graph used in the reachable and shortest path example.

If the syntax tree contains a deterministic result (line 4), we simply add this result to the environment. If the syntax contains a failure (line 5), the environment remains unchanged. When presented with a binary choice (line 6), we first update the environment according to the left branch and then according to the right branch. Finally, in the case of a recursive call $Rec\ j\ k$, we first check if the map contains an entry for the argument j (line 7). If it has no such entry, we add an entry containing the empty set (line 8), otherwise we update the environment based on the existing elements (line 9). Inserting an empty entry for j, ensures that the next iteration of *step* also updates the map for $expr\ j$.

We still need a function *lfp* to iteratively compute a least fixed point. Given an initial value, this function keeps calling itself until the result no longer changes.

$$\begin{aligned}&lfp :: Eq\ a \Rightarrow (a \to a) \to a \to a\\&lfp\ f\ a_0 = \mathbf{let}\ a_1 = f\ a_0\ \mathbf{in\ if}\ a_1 \equiv a_0\ \mathbf{then}\ a_1\ \mathbf{else}\ lfp\ f\ a_1\end{aligned}$$

As an example, consider the semantics of $pairNDRec$:

$$\begin{aligned}&\ggg runNDRec\ pairNDRec\ ()\\&fromList\ [(1,2),(2,1)]\end{aligned}$$

The denotational semantics and its Haskell implementation coincide as expected, but only up to termination (as we show in Section 6).

Graph Example We compute reachability in a cyclic directed graph using our non-determinism framework. We use the adjacency list representation of a graph where every $Node$ has a label (of type $String$) and a list of adjacent nodes:

$$\begin{aligned}&\mathbf{data}\ Node = Node\ \{label :: String, adj :: [Node]\}\\&\mathbf{instance}\ Eq\ \ Node\ \mathbf{where}\ n_1 \equiv n_2 = label\ n_1 \equiv label\ n_2\\&\mathbf{instance}\ Ord\ Node\ \mathbf{where}\ n_1 \leqslant n_2 = label\ n_1 \leqslant label\ n_2\end{aligned}$$

Our example graph (see Figure 3) consists of five nodes. It contains two cycles: from 3 to 4 and back, and from 5 to itself.

$$\begin{aligned}[n_1,n_2,n_3,n_4,n_5] = [&Node\ \texttt{"1"}\ [n_2,n_5],\\&Node\ \texttt{"2"}\ [n_3],\\&Node\ \texttt{"3"}\ [n_4],\\&Node\ \texttt{"4"}\ [n_3,n_1],\\&Node\ \texttt{"5"}\ [n_5]]\end{aligned}$$

Reachability is straightforwardly computed as:

$$\begin{aligned}&reach :: Node \to NDRec\ Node\ Node\ Node\\&reach\ n = return\ n\ `Or`\ (choose\ (adj\ n) \ggg\!= rec)\end{aligned}$$

i.e. n is reachable from n and all nodes reachable from a neighbor of n are reachable from n.

$$\begin{aligned}&\ggg runNDRec\ reach\ n_1\\&fromList\ [Node\ 1, Node\ 2, Node\ 3, Node\ 4, Node\ 5]\end{aligned}$$

2: ∅,[]		
2: ∅,[]	1: ∅,$[go\ 2 \circ k_0]$	
2: ∅,[]	1: {1},$[go\ 2 \circ k_0]$	
2: ∅,[]	1: {1},$[go\ 2 \circ k_0]$	0: ∅,$[go\ 1 \circ l_1]$
2: ∅,[]	1: {1},$[go\ 2 \circ k_0]$	0: {0},$[go\ 1 \circ l_1]$
2: {2},[]	1: {1},$[go\ 2 \circ k_0]$	0: {0},$[go\ 1 \circ l_1]$
2: {2},[]	1: {1},$[go\ 2 \circ k_0]$	0: {0},$[go\ 1 \circ l_1]$

Table 1: The trace of the environment for $runNDk\ fib\ 2$.

5. DEPENDENCY TRACKING

The effect handler that was provided in Section 4 computes the least fixed point in a very naive way, resulting in sub par performance. To illustrate the problem, consider the following program to compute the Fibonacci numbers:

$$\begin{aligned}&fib :: Int \to NDRec\ Int\ Int\ Int\\&fib\ n \mid n \equiv 0 \quad = return\ 0\\&\quad\ \mid n \equiv 1 \quad = return\ 1\\&\quad\ \mid otherwise = \mathbf{do}\ f_1 \leftarrow rec\ (n-1)\\&\qquad\qquad\qquad\ \ f_2 \leftarrow rec\ (n-2)\\&\qquad\qquad\qquad\ \ return\ (f_1 + f_2)\end{aligned}$$

The evolution of the environment for $runNDRec\ fib\ 2$ after each application of *step* is shown in Table 1 (ignore the list after the comma for now). Every value in the environment is recomputed in every step. The work performed by the effect handler is monotonic: the work performed by a single iteration is also performed in all later iterations. In the case of *fib* this leads to an $\mathcal{O}(n^2)$ runtime.[8]

This duplication of work has two sources:

1. The naive least fixed point computation *lfp* iteratively applies the *step* function which folds over *all* keys in the environment, even when the set of outcomes of that key has not changed.
2. The function *go* computes too much: the only part of the syntax tree that is influenced by a recursive call $Rec\ j\ k$ is the *continuation* k of that recursive call. Therefore only the continuation needs to be recomputed.

Although efficiency is not the main focus of this paper, we would be remiss not to address such a glaring problem. Especially since the solution is simple: both problems can be solved by keeping track of which continuation depends on which recursive call (identified by its argument), and only evaluating the continuation *once* for every new value of the recursive call.

Figure 4 shows the code for the interpreter $runNDk$ that tracks dependencies. This effect handler relies on seven auxiliary functions defined for the type $Env\ i\ o$. This type abstracts over environments that contain continuations in addition to a set of values. We show their signatures and implementations in Figure 4.

The computation starts with the call to *go* on line 1, which returns the environment after the least fixed point computed. From this environment the result is extracted using !!!. The *go* function proceeds by case analysis: if the syntax tree is $Success\ x$ and the value x is not yet in the environment, the environment is updated by sequentially applying every continuation that is currently in the environment (line 2-3). If the syntax tree is a failed computation the environment is unchanged (line 4). If the syntax tree is a binary choice (line 5) the environment is updated first for the left

[8]modulo a logarithmic factor for the use of finite maps.

branch, then for the right branch. If the syntax tree is a recursive call $Rec\ j\ k$ (line 6-10), and there is no entry for j in the map (line 9), i.e. this is the first time we see a call to j, a continuation for j is added (line 7) and the recursive function is evaluated for j. Otherwise (line 10), a new continuation is added for j, and this continuation is applied to every known result of j. The least fixed point is reached when the $ifNew$ check on line 2 no longer succeeds.

The trace for $runNDk\ fib\ 2$ is identical to the trace for $runND\ fib\ 2$, but now only a single entry is updated in each iteration. Moreover, the result from the call to 1 only triggers the continuation $k_0 = \lambda f_1 \to Rec\ 0\ (\lambda f_2 \to return\ (f_1 + f_2))$ and the result of 0 only triggers computation of $l_1 = \lambda f_2 \to return\ (1 + f_2)$, instead of the entire syntax tree *fib* 2.

In summary, we obtain a linear runtime for *fib*.

6. LATTICES

Reconsider the graph defined in Section 4. Instead of just finding all reachable nodes, we may want to additionally know the length of the *shortest* path. The program *sp* is a first attempt at computing the shortest path between *src* and *dst*.

$$
\begin{array}{l}
sp :: Node \to Node \to NDRec\ Node\ Int\ Int \\
sp\ dst\ src \\
\quad \mid dst \equiv src\ = return\ 0 \\
\quad \mid otherwise = choose\ (adj\ src) \gg\!= rec \gg\!= return \circ (+1)
\end{array}
$$

The denotational semantics gives us the following value for the expression $sp\ n_1\ rec\ n_2$:

$$[\![\ sp\ n_1\ rec\ ]\!](n_2) = \{2, 4, 6, 8, \ldots\}$$

Since this value is infinite, the effect handler does not terminate. In order to implement a terminating effect handler, we need to first define an alternative denotational semantics that at least produces a finite value, which we can then compute in a finite amount of time.

Lattice-based Semantics Instead of powersets $\mathcal{P}(O)$, we generalize the semantic domain to an arbitrary complete lattice L. On this domain we define a new denotational semantics for non-deterministic computations. The signature of this denotational semantics is:

$$[\![\cdot]\!]_L : (I \to NDRec\ I\ L\ L) \to (I \to L)$$

where $\langle L, \sqsubseteq \rangle$ is a complete lattice.

A complete lattice is a partially ordered set $\langle S, \sqsubseteq \rangle$ such that for every subset $X \in S$ there exists a least upper bound, i.e. an element $\sqcup X$ that is the least element that is larger than every element in X, more formally:

DEFINITION 1 (COMPLETE LATTICE). *A complete lattice $\langle L, \sqcup \rangle$ is a partially ordered set $\langle L, \sqsubseteq \rangle$ where $\forall X \subseteq L$ there exists a least upper bound $\sqcup X \in L$, such that:*

$$\forall z \in L : \sqcup X \sqsubseteq z \iff \forall x \in X : x \sqsubseteq z$$

It follows from the definition that $\sqcup X$ is unique. A complete lattice L always has a least element $\bot$ that is smaller than all other elements. It corresponds to the least upper bound of the empty set: $\bot = \sqcup\emptyset$. With a slight abuse of notation we also write $a \sqcup b$ to denote the least upper bound $\sqcup\{a, b\}$, pronounced "join".

As before, we first consider the semantics for the case where we already possess an environment $s : I \to L$ containing the (partial) solution for every call $f(a)$, i.e.

$$s(a) \sqsubseteq [\![\ f\ ]\!]_L(a) \text{ for every } a \in I$$

Then for every syntax tree $t : NDRec\ I\ L\ L$ we can define the denotational semantics $\mathcal{R}[\![\cdot]\!]_L$ in the new setting:

$$
\begin{array}{ll}
\mathcal{R}[\![\cdot]\!]_L\text{: } NDRec\ I\ L\ L \to (I \to L) \to L & \\
\mathcal{R}[\![\ Success\ x &]\!]_L(s) = x \\
\mathcal{R}[\![\ Fail &]\!]_L(s) = \bot \\
\mathcal{R}[\![\ Or\ l\ r &]\!]_L(s) = \mathcal{R}[\![\ l\ ]\!]_L(s) \sqcup \mathcal{R}[\![\ r\ ]\!]_L(s) \\
\mathcal{R}[\![\ Rec\ j\ k &]\!]_L(s) = \mathcal{R}[\![\ k(s(j))\ ]\!]_L(s)
\end{array}
$$

This semantics is very similar to the previously defined semantics for sets. In particular, the semantics $\mathcal{R}[\![\cdot]\!]$ almost corresponds to the semantics $\mathcal{R}[\![\cdot]\!]_L$ specialized to the powerset of the output type $(\mathcal{R}[\![\cdot]\!]_{\mathcal{P}(O)})$. We will make this relationship more precise in Section 6.1.

For $[\![\ f\ ]\!]_L$ we would again like to compute the least fixed point of $\lambda s.\lambda a.\big(\mathcal{R}[\![\ f(a)\ ]\!]_L(s)\big)$. However, unlike for sets, this function is not guaranteed to be monotonic for arbitrary lattices in general, so a least fixed point might not always exist. Instead, we define the operator $\Uparrow$:

DEFINITION 2. *The operator $\Uparrow$ is defined as:*

$$
\begin{aligned}
f \Uparrow 0 &= \bot \\
f \Uparrow (i+1) &= f \Uparrow i \sqcup f(f \Uparrow i) \\
f \Uparrow \infty &= \bigsqcup_{i=0}^{\infty} (f \Uparrow i)
\end{aligned}
$$

Observe that the sequence $f \Uparrow 0$, $f \Uparrow 1$, $f \Uparrow 2, \ldots$, $f \Uparrow \infty$ is non-decreasing.[9]

If the domain of f has the *finite ascending chain property* this sequence is finite, and then $f \Uparrow \infty$ must be a fixed point. Thus, we define $[\![\cdot]\!]_L$ as:

$$[\![\ f\ ]\!]_L = (\lambda s.\lambda a.\mathcal{R}[\![\ f(a)\ ]\!]_L(s)) \Uparrow \infty$$

Moreover, if $\lambda s.\lambda a.\mathcal{R}[\![\ f(a)\ ]\!]_L(s)$ is continuous, then $\Uparrow$ computes the least fixed point, i.e.

$$lfp(\lambda s.\lambda a.\mathcal{R}[\![\ f(a)\ ]\!]_L(s)) = (\lambda s.\lambda a.\mathcal{R}[\![\ f(a)\ ]\!]_L(s)) \Uparrow \infty = [\![\ f\ ]\!]_L$$

Otherwise, $\Uparrow$ may compute a fixed point that is strictly greater than the least fixed point. In practice, however, almost all functions are continuous.

Shortest Path Example For our shortest path problem, the relevant partial order is $\langle \mathbb{N} \cup \{\infty\}, \sqsubseteq \rangle$ where

$$a \sqsubseteq b \iff a = \infty \text{ or } b \leqslant a$$

which is the reverse order of the canonical order on $\mathbb{N}$, i.e. smaller (shorter) is better. Since the canonical order is well-founded (i.e. has no infinite descending chains), $\sqsubseteq$ has the finite ascending chain property. The relevant lattice is $\langle \mathbb{N} \cup \{\infty\}, \sqcup \rangle$ where

$$\sqcup X = \begin{cases} \infty & \text{if } X = \emptyset \\ \min_{\sqsubseteq} X & \text{otherwise} \end{cases}$$

In this lattice the least fixed point exists, since continuation $return \circ (+1)$ in $sp\ n_1$ is continuous.

$$[\![\ sp\ n_1\ ]\!]_{\mathbb{N}\cup\{\infty\}}(n_2) = 3$$

[9]For environments, assume the order $\preccurlyeq$ defined as:

$$s_1 \preccurlyeq s_2 \iff \forall a \in I : s_1(a) \sqsubseteq s_2(a)$$

```
runNDk :: (Ord i, Ord o) ⇒ (i → NDRec i o o) → i → Set o
runNDk expr i₀ = go i₀ (expr i₀) emptyEnv !!! i₀ where          – (1)
  go i (Success x) m = ifNew i x m newEnv where                 – (2)
    newEnv = foldr ($x) (store i x m) (conts i m)               – (3)
  go i Fail        m = m                                        – (4)
  go i (Or l r)    m = go i r (go i l m)                        – (5)
  go i (Rec j k)   m =                                          – (6)
    let newEnv = addCont j (go i ∘ k) m                         – (7)
    in case results j m of                                      – (8)
      Nothing → go j (expr j) newEnv                            – (9)
      Just rs → foldr (go i ∘ k) newEnv (toList rs)             – (10)

ifNew :: (Ord i, Ord o)
      ⇒ i → o → Env i o → Env i o → Env i o
ifNew i o env newEnv =
  if maybe True (¬ ∘ member o) (results i env)
    then newEnv
    else env

type Env i o = M.Map i (Set o, [C i o])
newtype C i o = C {runC :: o → Env i o → Env i o}

emptyEnv :: Env i o
emptyEnv = M.empty

store :: (Ord i, Ord o) ⇒ i → o → Env i o → Env i o
store i o = M.insertWith mappend i (singleton o, [])

results :: Ord i ⇒ i → Env i o → Maybe (Set o)
results i = fmap fst ∘ M.lookup i

(!!!) :: Ord i ⇒ Env i o → i → Set o
m !!! i = fromJust (results i m)

conts :: Ord i ⇒ i → Env i o → [o → Env i o → Env i o]
conts i = map runC ∘ M.findWithDefault [] i ∘ fmap snd

addCont :: (Ord i, Ord o) ⇒ i → (o → Env i o → Env i o)
        → Env i o → Env i o
addCont i k = M.insertWith mappend i (empty, [C k])
```

Figure 4: Effect Handler and Environment type for dependency tracking.

The length of the shortest path from node 2 to node 1 is indeed 3. When a path does not exist, we get the following length:

$$[\![\ sp\ n_1\ ]\!]_{\mathbb{N}\cup\{\infty\}}(n_5) = \infty$$

Notice the similarity to solving a problem with *Dynamic Programming*. The idea of dynamic programming is to compute the optimal solution to a problem by combining optimal solutions to smaller instances of the same problem. In this setting, recursive calls correspond to instances of the same problem. The lattice allows us to keep only the optimal solution to a given instance, from which the result of a larger instance is computed.

Subset Sum Example Another example is the following: *Given a list of integer numbers and an integer s, find the shortest non-empty sublist that sums to s.*

To solve this problem, first observe that lists form a lattice when ordered by descending length, if we adjoin a bottom element *InfList* representing any list of infinite length. This is embodied by the *Shortest*-datatype (see Section 4.2 for the definition of *Lattice*).

```
data Shortest = InfList | List [Int]

cons :: Int → Shortest → Shortest
cons n InfList    = InfList
cons n (List ns) = List (n : ns)

instance Lattice Shortest where
  bottom = InfList
  join InfList a = a
  join a InfList = a
  join (List a) (List b)
    | length a ⩽ length b = List a
    | otherwise           = List b
```

The function *sss* returns the *Shortest* list which sums to its first argument and is a sublist of of its second argument.

```
sss :: (Int, [Int]) → NDRec (Int, [Int]) Shortest Shortest
sss (n, [])
  | n ≡ 0      = return (List [])
  | otherwise = Fail
sss (n, x : xs) = choice [rec (n, xs),
                          fmap (cons x) (rec (n − x, xs))]
```

When the input list is empty we can only construct an empty list with sum 0. When the input list is non-empty, the shortest list summing to n is obtained by either recursively searching for a list summing to n, or *cons*-ing x onto a list that sums to $n - x$.

The function $\mathcal{R}[\![\ sss\ \ (n, xs)\ ]\!]_L$ is continuous for all $(n,\ xs)$. Moreover, the environment possesses a finite ascending chain property: every recursive call decreases the size of the input list by one. Then, given a finite input list, there can only be a finitely many recursive calls.

For instance, consider the application of *sss* to $(10, [5, 0, 5])$:

$$[\![\ sss\ ]\!](10, [5, 0, 5]) = List[5, 5]$$

Indeed $sum\ [5, 5] \equiv 10$. Note that the list $[5, 0, 5]$ itself also sums to 10. However, only the shortest list is retained.

Grammar Analysis example We demonstrate the added power of the lattice-based semantics by solving a simple grammar analysis problem. Informally, a grammar is a list of rules or productions:

```
type Grammar = [Production]
```

A production is of the form

```
data Production = Symbol ⟼ [Symbol]
```

There is a single *Symbol* to the left of $\longmapsto$, called the head, and zero or more symbols, the body, on the right hand side. Symbols can be either terminals or non-terminals:

```
type Symbol = Char
isTerminal :: Symbol → Bool
isTerminal s = ¬ (isUpper s)
```

Terminals may not appear in the head of a rule. For our purpose we distinguish terminals and non-terminals based on whether they are upper or lower case characters.

Now, consider the following grammar:

```
grammar :: Grammar
grammar = ['E' ⟼ "T Z",   'E' ⟼ "(E)",
           'Z' ⟼ "+ T Z", 'Z' ⟼ "+ (E)",
           'Z' ⟼ "",
           'T' ⟼ "a",     'T' ⟼ "1"]
```

This grammar describes a simple expression language consisting of one identifier (a), one literal (1) and a binary operator ($+$). We implement a program that analyses grammars for nullability: a symbol is *Nullable* if it derives the empty string. Nullability is charac-

terized by a function from *Symbol* to *Any*, where *Any* is a lattice of booleans, where *join* is disjunction and *False* is *bottom*.

```
newtype Any = Any {getAny :: Bool}
instance Lattice Any where
  bottom = Any False
  join (Any a) (Any b) = Any (a ∨ b)
```

Given this lattice, the implementation of *nullable* is straightforward:

```
nullability :: Grammar → Symbol → NDRec Symbol Any Any
nullability g s
  | isTerminal s = return (Any False)
  | otherwise = do
      head ⟼ body ← choose g
      guard (head ≡ s)
      nullables ← mapM rec body
      return (Any (and (map getAny nullables)))
```

A terminal symbol is never nullable. In order to decide if a non-terminal symbol s is nullable, we need to check if there exists a production with s in the head and with a body consisting of nullable symbols. Note that s may also occur in the body.

We evaluate *nullability* for *grammar*:

$$[\![\, nullability\ grammar \,]\!]_{Any}(\texttt{'T'}) = Any\ False$$
$$[\![\, nullability\ grammar \,]\!]_{Any}(\texttt{'Z'}) = Any\ True$$
$$[\![\, nullability\ grammar \,]\!]_{Any}(\texttt{'E'}) = Any\ False$$

We have demonstrated that a semantics based on lattices allows us to tackle problems we could not previously solve in a finite amount of time. Note that we again arrived at this solution by computing a least fixed point, but of a different function. Furthermore, only the semantics of the syntax has changed, the syntax itself is unmodified. Hence, to implement this semantics only the effect handler needs to be reworked.

6.1 Generalized Effect Handler

This section shows that the semantics from Section 6 is strictly more general than the semantics from Section 4.2. The *NDRec* monad is a free monad. More precisely, it is the coproduct monad of two other free monads respectively implementing non-determinism and recursion. In this section we make this coproduct explicit by following the data types à la carte approach [20]. All free monads have the following shape:

```
data Free f a = Return a | Free (f (Free f a))
```

They either return a pure value (*Return*) or an impure effect (*Free*), constructed using f. When f is a *Functor*, *Free f* is a monad:

```
instance Functor f ⇒ Monad (Free f) where
  return a = Return a
  Return a >>= f = f a
  Free g   >>= f = Free (fmap (>>=f) g)
```

Functions interpreting *Free* are built using the function *fold*:

```
fold :: Functor f ⇒ (a → b) → (f b → b) → Free f a → b
fold ret alg (Return a) = ret a
fold ret alg (Free f)   = alg (fmap (fold ret alg) f)
```

where the first argument *ret* is applied to pure values and the second argument *alg* is an algebra that is applied to impure values.

We use the coproduct $+$ of two *Functor*s to express the combination of two effects:

```
data (f + g) a = Inl (f a) | Inr (g a) deriving Functor
```

This data type is similar to the *Either* type, but it works on type constructors (kind $* \to *$) instead of types (kind $*$). We use the following two effect signatures to express recursion and non-determinism respectively. Their functor instances are automatically derived by GHC.

```
data RecF i o a = RecF i (o → a) deriving Functor
data NDF a = OrF a a | FailF     deriving Functor
```

For convenience, we label the combined free monad (which is isomorphic to the *NDRec* type) with a type synonym *NDRecF*, and define some smart constructors for it:

```
type NDRecF i o = Free (NDF + RecF i o)
orF :: NDRecF i o a → NDRecF i o a → NDRecF i o a
orF l r = Free (Inl (OrF l r))
chooseF :: [a] → NDRecF i o a
chooseF = foldr orF failF ∘ map return where
  failF = Free (Inl FailF)
recF :: i → NDRecF i o o
recF i = Free (Inr (RecF i Return))
```

Effect Handler Implementation for Lattices The typeclass *Lattice* is defined as follows:

```
class Lattice l where
  bottom :: l
  join   :: l → l → l
```

This typeclass prioritizes the view of the $\sqcup$-operator as a binary operator because this is the most useful in practice.

Now we replace every *FailF* in a value of type *NDRecF* with *bottom* and every *OrF* with *join*:

```
runND :: (Functor f, Lattice o)
      ⇒ Free (NDF + f) o → Free f o
runND = fold return alg where
  alg (Inl FailF)      = return bottom
  alg (Inl (OrF l r)) = join <$> l <*> r
  alg (Inr f)          = Free f
```

The effect handler *runND* effectively eliminates the *NDF* effect from the syntax tree. Only the recursive effect *RecF i o* remains. This effect is then interpreted by *runRec*:

```
runRec :: (Ord i, Eq o, Lattice o)
       ⇒ (i → Free (RecF i o) o) → i → o
runRec expr i₀ = lfp step s₀ ! i₀ where
  s₀      = M.singleton i₀ bottom
  step m  = foldr (λk → go (expr k) k) m (M.keys m)
  go expr = fold onReturn alg expr where
    onReturn x i = M.insertWith join i x
    alg (RecF j k) i m = case M.lookup j m of
      Nothing → k bottom i (M.insert j bottom m)
      Just l  → k l i m
```

Note that *runRec* does not expect a plain effect, but an effectful function $i \to$ *Free* (*RecF i o*) o, and also returns a function $i \to o$. Essentially, *runRec* is a memoizing fixpoint operator applied to the recursive equation defined by *expr*.

Reconsider the shortest path program from Section 6:

```
spF :: Node → Node → NDRecF Node Dist Dist
spF dst src | src ≡ dst = return 0
            | otherwise = do n ← chooseF (adj src)
                             d ← recF n
                             return (d + 1)
```

where the $Dist$ datatype and its instances are defined as:

```
data Dist = InfDist | Dist Int deriving (Eq, Show)
instance Lattice Dist where
  bottom = InfDist
  join InfDist   a         = a
  join a         InfDist   = a
  join (Dist d1) (Dist d2) = Dist (min d1 d2)
instance Num Dist where
  InfDist + a         = InfDist
  a       + InfDist   = InfDist
  Dist d1 + Dist d2   = Dist (d1 + d2)
  fromInteger = Dist ∘ fromInteger
```

Note that $Dist$ has a $Lattice$-instance corresponding to the lattice $(\mathbb{N} \cup \{\infty\}, \sqcup)$ from Section 6. Combining $runND$ and $runRec$ gives an effect handler $runRec \circ (runND \circ)$ implementing the semantic function $[\![\cdot]\!]_L$. For spF this effect handler computes the expected result:

```
>>> runRec (runND ∘ spF n1) n2
Dist 2
>>> runRec (runND ∘ spF n1) n5
InfDist
```

Lifting Continuations The semantic functions $\mathcal{R}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]_L$ only differ in the case for $Rec\ j\ k$: $\mathcal{R}[\![\cdot]\!]$ computes the union of the k applied to every element of $s(j)$, while $\mathcal{R}[\![\cdot]\!]_L$ directly applies the k to $s(j)$. The similarity between $\mathcal{R}[\![\cdot]\!]$ and $\mathcal{R}[\![\cdot]\!]_{\mathcal{P}(O)}$ suggests that we can obtain the behaviour of the former with the latter if we just change the continuation in $Rec\ j\ k$ appropriately. More formally, we want to construct a continuation k' such that:

$$\mathcal{R}[\![\ Rec\ j\ k\ ]\!](s) = \mathcal{R}[\![\ Rec\ j\ k'\ ]\!]_{\mathcal{P}(O)}(s)$$

This is achieved by the following code, that lifts a computation in the $Free\ (NDF + RecF\ i\ o)$-monad to one in the $Free\ (NDF + Rec\ i\ (Set\ o))$-monad.

```
liftRec :: Ord o
        ⇒ Free (NDF + RecF i   o       ) o
        → Free (NDF + RecF i (Set o)) (Set o)
liftRec = fold (return ∘ singleton) alg where
  alg (Inl f)             = Free (Inl f)
  alg (Inr (RecF i k)) = Free (Inr (RecF i k')) where
    k' = fmap unions ∘ traverse k ∘ toList
```

Since the powerset $\langle \mathcal{P}(O), \cup \rangle$ forms a lattice, we can define a $Lattice$ instance for Set:

```
instance Ord a ⇒ Lattice (Set a) where
  bottom = empty
  join   = union
```

Composing $liftRec$, $runND$ and $runRec$ yields a complete effect handler for $Free\ (NDF + RecF\ i\ o)$, corresponding to $[\![\cdot]\!]$:

```
runNDRecF :: (Ord i, Ord o)
          ⇒ (i → Free (NDF + RecF i o) o)
          → (i → Set o)
runNDRecF f = runRec (runND ∘ liftRec ∘ f)
```

Examples Redefining our running example:

```
pairF :: () → NDRecF () (Int, Int) (Int, Int)
pairF () = return (1,2) `orF` do (x, y) ← recF ()
                                 return (y, x)
```

leads to the following familiar result:

```
>>> runRecF pairF ()
fromList [(1,2),(2,1)]
```

To recapitulate, we have shown how to implement the effect handler for $[\![\cdot]\!]_L$ for $NDRecF$. On top of this we implement the effect handler for $[\![\cdot]\!]$ for sets, showing that the semantics for lattices generalize the semantics for sets.

7. MUTUAL RECURSION

The syntax we have defined so far has a serious limitation: it can only handle a single function at a time. This section lifts this limitation by extending the syntax to support several, potentially mutually recursive functions within the same non-deterministic computation. Moreover, these functions are allowed to possess different argument and return types.

Motivating Example We demonstrate the added power with another grammar analysis. We implement a program that finds the $First$ set of a symbol. A symbol X is in the $First$ set of another symbol Y if any of the strings derived from Y start with X. Solving $First$ requires also solving $Nullable$.

Recall that nullability is characterized by a function from $Symbol$ to Any. First is a also characterized by a function, but one from $Symbol$ to $Set\ Symbol$ (recall that Set is also a lattice). We encode this fact with a Generalized Algebraic Data Type (GADT) [14]:

```
data Analyze o where
  Nullable :: Symbol → Analyze Any
  First    :: Symbol → Analyze (Set Symbol)
```

The arguments to the *data* constructors $Nullable$ and $First$ indicate the argument type (in both cases $Symbol$). The argument to the *type* constructor $Analyze$ indicates the return type (Any or $Set\ Symbol$).

The syntax data type NDM is a variation of $NDRec$:

```
data NDM i a where
  SuccessM :: a → NDM i a
  FailM    :: NDM i a
  OrM      :: NDM i a → NDM i a → NDM i a
  RecM     :: Lattice o
           ⇒ i o → (o → NDM i a) → NDM i a
```

The recursive call constructor $RecM$ expects an existentially qualified parameter o (which must be a lattice). The previous syntax introduces o in the type, and as such fixes o to one particular type for the entire syntax tree. Because of the existential parameter, NDM *does* allow calls with different return types. Furthermore, the argument is of type $i\ o$. When i is a GADT, such as $Analyze$, the type $i\ o$ will only be inhabited for the desired os (e.g. Any and $Set\ Symbol$ for $Analyze$).

We also define four additional smart constructors $recM$, $chooseM$, $choiceM$ and $guardM$. Their meaning is analogous to the smart constructors for $NDRec$. For brevity, their signature and implementation is omitted here. With these smart constructors we define explicit recursive calls for $nullable$ and $first$.

```
nullable :: Symbol → NDM Analyze Bool
nullable = fmap getAny ∘ recM ∘ Nullable

first :: Symbol → NDM Analyze Symbol
first s = recM (First s) >>= chooseM ∘ toList
```

Solving $Nullable$ and $First$ is straightforward:

```
analyze :: Grammar → Analyze o → NDM Analyze o
analyze g (Nullable s)
```

```
  | isTerminal s = return (Any False)
  | otherwise = do
      head ⟼ body ← chooseM g
      guardM (head ≡ s)
      nullables ← mapM nullable body
      return (Any (and nullables))
analyze g (First s)
  | isTerminal s = return (singleton s)
  | otherwise = do
      head ⟼ body ← chooseM g
      guardM (head ≡ s)
      nullables ← mapM nullable body
      let nulls     = takeWhile snd (zip body nullables)
          notNulls = dropWhile snd (zip body nullables)
          prefix    = nulls ++ take 1 notNulls
          firsts    = map (first ∘ fst) prefix
      terminal ← choiceM firsts
      return (singleton terminal)
```

In order to determine the *First* set of a symbol s, we need to find the longest nullable prefix of the body for every production with s in the head. Then we find the *First* sets of every symbol in the prefix and the symbol directly following the prefix.

Implementation From *Analyze* we derive the actual input and output types that need to be stored in the environment:

```
type EnvM i = M.Map (Input i) (Output i)
data Input (i :: * → *) where
  Input :: Lattice o ⇒ i o → Input i
data Output (i :: * → *) where
  Output :: i o → o → Output i
```

The data types *Input* and *Output* wrap a particular instantiation of the *Analyze* type constructor in an existential quantification. The implementation of the *Eq* and *Ord* instances is elided for brevity.

We use the typeclass *Coerce* to type-safely coerce *Output Analyze* back to *Any* or *Set Symbol*, depending on which constructor of *Analyze* is pattern-matched.

```
class Coerce c where
  coerce :: c o → Output c → Maybe o
instance Coerce Analyze where
  coerce (Nullable s) (Output (Nullable t) o) = Just o
  coerce (First s)    (Output (First t) o)    = Just o
  coerce a            o                       = Nothing
```

Now we lift *join* to *Output*s:

```
joinO :: (Coerce i, Lattice o)
      ⇒ i o → Output i → Output i → Output i
joinO i o1 o2 =
  maybe o1 (Output i) (join <$> coerce i o1
                            <*> coerce i o2)
```

At last we have all the pieces to define the effect handler. The function *goM* evaluates an *NDM i o*, given an environment *EnvM i* and *runNDM* computes the least fixed point of *goM*:

```
goM :: (Coerce i, Lattice o, Ord (Input i))
    ⇒ i o → NDM i o → EnvM i → EnvM i
goM i (SuccessM x) m =
  M.insertWith (joinO i) (Input i) (Output i x) m
goM i FailM        m = m
goM i (OrM l r)    m = goM i r (goM i l m)
goM i (RecM j k)   m =
  case M.lookup (Input j) m ≫= coerce j of
    Nothing → M.insert (Input j) (Output j bottom) m
    Just s  → goM i (k s) m
runNDM
  :: (Ord (Input i), Eq (Output i), Coerce i, Lattice a)
  ⇒ (∀ o . Lattice o ⇒ i o → NDM i o) → i a → a
runNDM expr i0 =
  fromJust (coerce i0 (lfp step s0 ! Input i0)) where
    s0 = M.singleton (Input i0) (Output i0 bottom)
    step m = foldr goM' m (M.keys m) where
      goM' (Input k) = goM k (expr k)
```

This implementation is not very different from *runNDRec* (see Section 4), except that some wrapping and unwrapping of *Input* and *Output* is required. Note that dependency tracking is orthogonal to the problem of mutual recursion, i.e. dependency tracking is easily added.

Evaluation Using *runNDM* we now solve *First*, which requires solving *Nullable*, *in the same computation*.

```
analyzeF :: Grammar → Symbol → [Symbol]
analyzeF g = toList ∘ runNDM (analyze g) ∘ First

≫≫ map (analyzeF grammar) ['E','Z','T']
["(1a","+","1a"]
```

The computed *First* sets of `E`,`Z` and `T` are the results one would expect from the definition: an expression may start with a parenthesis, a literal or an identifier, and similarly for sub-expressions and terms. We now extend *First* to strings of symbols:

```
firstS :: Grammar → [Symbol] → Set Symbol
firstS g s =
  let nulls     = takeWhile analyzeN s
      notNulls = dropWhile analyzeN s
      prefix    = nulls ++ take 1 notNulls
      analyzeN = getAny ∘ runNDM (analyze g) ∘ Nullable
  in unions $ map (runNDM (analyze g) ∘ First) prefix
```

So called *left-recursive conflicts* occur when the *First* sets of bodies of two rules with the same head have a non-empty intersection. The following function *hasConflict* uses *firstS* of rule bodies to detect conflicts:

```
hasConflict :: Grammar → Bool
hasConflict g =
  let ruleHead (h ⟼ _) = h
      ruleBody (_ ⟼ b) = b
      ruleConflict [r] = False
      ruleConflict rs  = ¬ (null (foldr1 intersection firsts))
        where firsts = map (firstS g ∘ ruleBody) rs
  in or $ map ruleConflict $ groupBy ((≡) `on` ruleHead) g
```

Our example grammar is conflict free:

```
≫≫ hasConflict grammar
False
```

However, in the following grammar the rules for 'E' do conflict:

```
grammar2 :: Grammar
grammar2 = ['E' ⟼ "E + T", 'E' ⟼ "T",
            'T' ⟼ "a",      'T' ⟼ "1"]
≫≫ hasConflict grammar2
True
```

Note that the implementation of *firstS* and *hasConflict* is not very efficient: in between calls to *runNDM* the results are simply thrown away. We see two ways to resolve this: either have

	Size	Naive	Dependency Tracking
Fibonacci	800	455.7	13.77
	805	462.8	13.85
	810	469.3	13.97
Knapsack	10	155.9	8.185
	15	226.6	12.32
	20	309.0	16.04
NQueens	7	4.864	1.259
	9	152.9	26.79
	10	914.8	132.0
SCC	30	57.31	8.843
	33	104.9	10.67
	35	97.97	12.67
Shortest Path	8000	1306	676.6
	8500	1362	718.7
	9000	1451	775.1

Table 2: The benchmark results in milliseconds.

$runNDM$ return the environment $EnvM$ to directly perform lookups, or embed $firstS$ and $hasConflict$ in NDM as well. Such an embedding requires extending $Analyze$ with new constructors for $FirstS$ and $HasConflict$. This is left as an exercise to the reader. In a more generic setting, one could imagine using a "data types à la carte"-approach to solve this extension more satisfactorily.

8. BENCHMARKS

To demonstrate the performance impact of dependency tracking, we evaluate several benchmarks with and without dependency-tracking effect handlers and compare their results.

We consider five problems: computing (large) fibonacci numbers, (a variation of) the knapsack problem, the nqueens problem, computing strongly connected components in a directed graph and finding the shortest path in a directed graph.

The benchmarks were build using the Criterion-benchmark harness and compiled using GHC 7.10.2 (with optimization setting `-O2`) on a 64-bit Linux machine with an Intel Core i7-2600 @ 3.6 Ghz and 16 GB of RAM. All code and results are available in the the Bitbucket repository.[10]

Table 2 shows the ordinary least squares regression of all samples of the execution time. The R^2 goodness of fit was above 0.99 in all cases. All values are in milliseconds.

For *Fibonacci* the first column is the index of the fibonacci number that is computed. For *Knapsack* it specifies the number of items to choose from (the capacity of the knapsack is fixed to 200). For *NQueens*, it indicates the number of queens that need to be placed on the board. For *SCC* it is the number of nodes in the graph (the number of edges is fixed to half the number of possible edges). Finally, for *Shortest Path* it is the number of edges in the graph.

The results show that effect handlers with dependency tracking massively outperform the naive effect handlers.

9. RELATED WORK

There are various related works.

Least fixed point computations In their seminal work [17] Scott and Strachey have introduced the idea of denotational mathematical semantics. In particular, they already define the semantics of programming language syntax based on least fixed points.

Jeannin et al. [8] give a category theoretic treatment of "Non-well-Founded computations". They give many examples, several of which can be implemented in our non-deterministic framework. Others are beyond our iterative least fixed point solver, and require more sophisticated techniques, for instance, Gaussian elimination.

The use of least fixed points for solving grammar analysis problems has been extensively treated by Jeuring and Swierstra [9].

It is a well-known result that dynamic programming can be viewed as memoization of recursive equations. The related work section of Swadi et al. gives an overview [19].

Effect Handlers Algebraic effects as a way to write effectful programs are the subject of much contemporary research [27, 10].

New languages have been created specifically for studying algebraic effect handlers: *Eff* [1] extends ML with syntax constructs for effect handlers and *Frank* [12] which has a Haskell-like syntax and tracks effect in its type system.

Non-determinism The classical introduction of non-determinism in functional programming based on the list monad is due to Wadler [24]. Hinze [7] derives a backtracking monad transformer from a declarative specification. This transformer adds backtracking (non-determinism and failure) to any monad.

Kiselyov et al. [11] improve on Hinze's work by implementing backtracking monad transformers which support fair non-deterministic choice. They present three different implementations: the classical approach based on *Streams* (i.e. lazy lists), one based on continuation passing style and another one based on control operators for delimited continuations.

Tabling in Prolog Since Prolog has native non-determinism, it too suffers from the issues we discussed in the introduction: non-determinism and recursion can interact to cause non-termination even when a finite solution does exist. The semantics of definite logic programs is given by the least fixed point of the *immediate consequence operator* T_P [23] that strongly resembles the semantics given by $[\![\cdot]\!]$. *Tabling* is a technique that produces a different semantics in the same way we produce different semantics for non-determinism. Several Prolog implementations support tabling, such as XSB-Prolog [22, 26], B-Prolog [28], Yap-Prolog [15], ALS-Prolog [5] and Mercury [18]. XSB-Prolog in particular also implements lattice-based answer subsumption [21] which strongly resembles and has partially inspired the techniques of Section 6. Other systems provide similar functionality through *mode-directed tabling* [16, 29, 6].

Explicit Recursion We are not the first to tame unbridled recursion by representing recursive calls explicitly. Devriese and Piessens [3, 4] overcome the limitations of so-called "ω-regular" grammars by representing recursive occurrences of non-terminals in production bodies explicitly.

McBride [13], working in the dependently typed total language Agda,[11] gives a free monadic model for recursive calls. Several possible semantics are discussed, but none based on fixed points.

Oliveira and Cook [2] extend traditional algebraic datatypes to with explicit definition and manipulation of cycles, offering a practical and convenient way of programming graphs in Haskell. Their approach to explicating recursion differs from ours. Where we use the smart constructor rec to indicate a recursive call, they instead use *open recursion*. While this is a good fit for the explicit graph data structure they work with, for our purposes (non-deterministic computations) the former style is more convenient.

10. CONCLUSIONS

We have described mutually recursive non-deterministic compu-

[10] https://bitbucket.org/AlexanderV/thesis, in the `IFL2015/benchmarks` directory.

[11] http://wiki.portal.chalmers.se/agda/pmwiki.php

tations in a free monadic fashion, and have given these descriptions a concise denotational semantics in terms of sets, and more generally in terms of lattices. We have shown how to efficiently implement these semantics in Haskell using Effect Handlers.

11. ACKNOWLEDGEMENTS

We thank Maciej Piròg for pointing out related work, and Georgios Karachalias, Steven Keuchel, Amr Saleh and Paolo Torrini, Maarten Van den Heuvel and the IFL 2015 participants for their helpful feedback. We are grateful to the anonymous reviewers of the IFL 2015 PC commitee for their useful comments. This research was partially supported by the Flemish Fund for Scientific Research (FWO). This work was partially supported by project GRACeFUL, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 640954.

12. REFERENCES

[1] A. Bauer and M. Pretnar. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108 – 123, 2015. Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

[2] B. C. d. S. Oliveira and W. R. Cook. Functional programming with structured graphs. In *ICFP*, pages 77–88. ACM, 2012.

[3] D. Devriese and F. Piessens. Explicitly recursive grammar combinators. In *Practical Aspects of Declarative Languages, PADL*, volume 11, pages 84–98. Springer, 2011.

[4] D. Devriese and F. Piessens. Finally tagless observable recursion for an abstract grammar model. *Journal of Functional Programming*, 22:757–796, 2012.

[5] H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings of the 17th International Conference on Logic Programming*, pages 181–196. Springer-Verlag, 2001.

[6] H.-F. Guo and G. Gupta. Simplifying dynamic programming via mode-directed tabling. *SoftwarePractice & Experience*, 38(1):75–94, 2008.

[7] R. Hinze. Deriving backtracking monad transformers. In *In The International Conference on Functional Programming (ICFP)*, 2000.

[8] J.-B. Jeannin, D. Kozen, and A. Silva. Language constructs for non-well-founded computation. In *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 61–80. Springer Berlin Heidelberg, 2013.

[9] J. Jeuring and D. Swierstra. Constructing functional programs for grammar analysis problems. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture*, pages 259–269. ACM, 1995.

[10] O. Kiselyov, A. Sabry, and C. Swords. Extensible Effects: An alternative to monad transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 59–70, New York, NY, USA, 2013. ACM.

[11] O. Kiselyov, C.-c. Shan, D. P. Friedman, and A. Sabry. Backtracking, interleaving, and terminating monad transformers:(functional pearl). In *ACM SIGPLAN Notices*, volume 40, pages 192–203. ACM, 2005.

[12] C. McBride. *The Frank Manual*, 2012.

[13] C. McBride. Turing-completeness totally free. In *Mathematics of Program Construction*, volume 9129 of *Lecture Notes in Computer Science*, pages 257–275. Springer International Publishing, 2015.

[14] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification based type inference for GADTs. In *In The International Conference on Functional Programming (ICFP)*, 2006.

[15] R. Rocha, F. Silva, and V. Santos Costa. YapTab: A tabling engine designed to support parallelism. In *Conference on Tabulation in Parsing and Deduction*, pages 77–87, 2000.

[16] J. Santos and R. Rocha. Efficient support for mode-directed tabling in the YapTab Tabling System. *CICLOPS 2012*, page 41, 2012.

[17] D. Scott and C. Strachey. *Toward a mathematical semantics for computer languages*, volume 1. 1971.

[18] Z. Somogyi and K. Sagonas. Tabling in Mercury: design and implementation. In *Proceedings of the 8th international conference on Practical Aspects of Declarative Languages*, pages 150–167. Springer-Verlag, 2006.

[19] K. Swadi, W. Taha, O. Kiselyov, and E. Pasalic. A monadic approach for avoiding code duplication when staging memoized functions. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 160–169, New York, NY, USA, 2006. ACM.

[20] W. Swierstra. Data types à la carte. *Journal of functional programming*, 18(04):423–436, 2008.

[21] T. Swift and D. S. Warren. Tabling with answer subsumption: Implementation, applications and performance. In *Logics in Artificial Intelligence*, pages 300–312. Springer, 2010.

[22] T. Swift and D. S. Warren. XSB: Extending Prolog with tabled logic programming. *Theory and Practice of Logic Programming*, 12(1-2):157–187, 2012.

[23] M. H. Van Emden and R. A. Kowalski. The Semantics of Predicate Logic As a Programming Language. *J. ACM*, 23(4):733–742, Oct. 1976.

[24] P. Wadler. How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

[25] P. Wadler. Monads for functional programming. In *Advanced Functional Programming*, pages 24–52. Springer, 1995.

[26] D. S. Warren. *Programming in Tabled Prolog*, volume 1. http://www3.cs.stonybrook.edu/~warren/xsbbook/, 1999.

[27] N. Wu and T. Schrijvers. Fusion for free - efficient algebraic effect handlers. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 302–322, 2015.

[28] N.-F. Zhou. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, 2012.

[29] N.-f. Zhou, Y. Kameya, and T. Sato. Mode-directed tabling for dynamic programming, machine learning, and constraint solving. In *Proceedings of the International Conference on Tools with Artificial Intelligence (ICTAI)*, 2010.